Zübeyir Bodur
21702382

CS202-1 HOMEWORK#2

Q1) Expression Tree

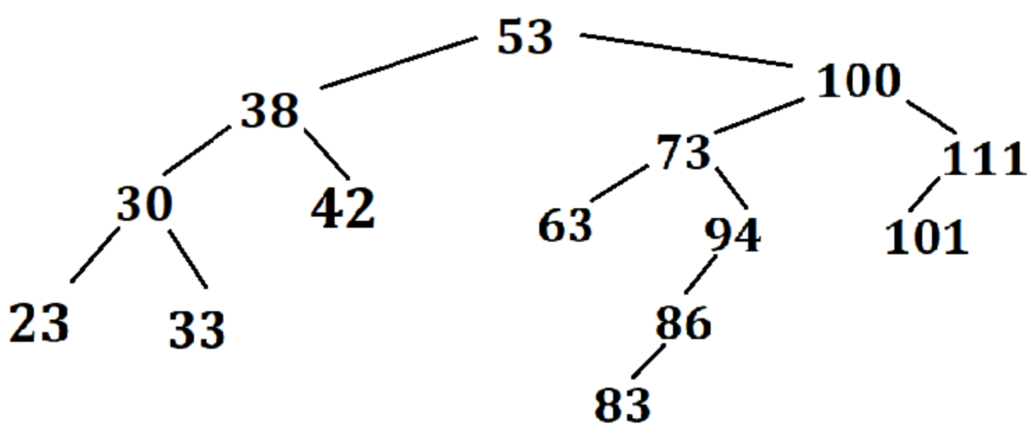
Prefix: A × B × [(C+D) - E] - [F / (G+H)]

Postfix: - × × A B - + C D E / F + G H

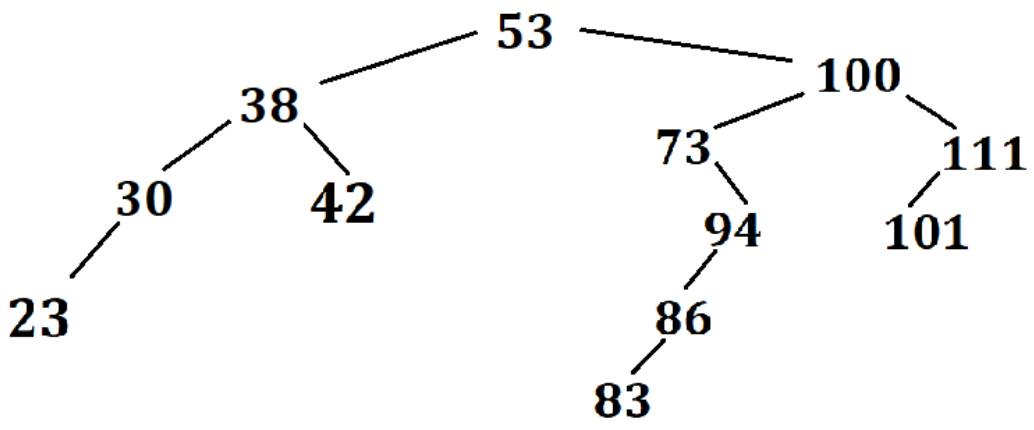
Infix: A B × C D + E - × F G H + / -

Q2) Drawing BST

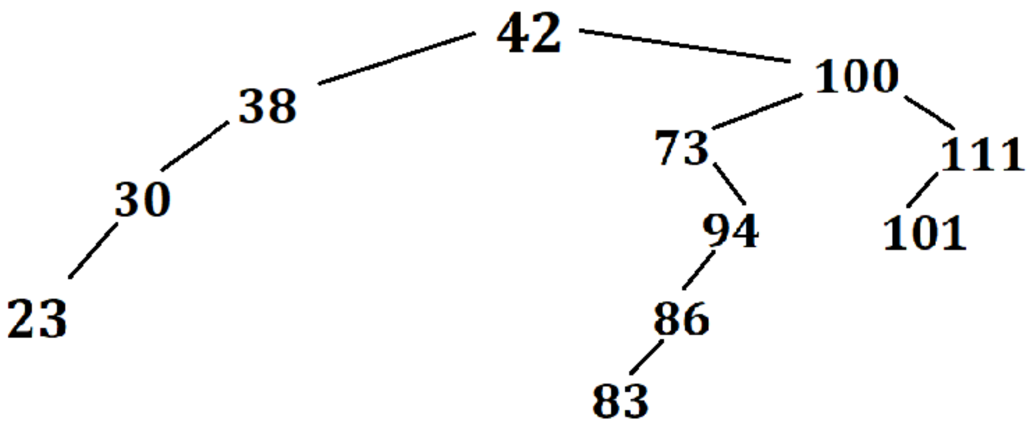
1. After all Insert Operations



2. After delete 63, 33



3. After delete 53



Q3) Sample Output

// input.txt
The height of a binary search tree

// Sample Output

Total 4-gram count: 10
"arch" appears 1 time(s)
"bina" appears 1 time(s)
"earc" appears 1 time(s)
"eigh" appears 1 time(s)
"heig" appears 1 time(s)
"ight" appears 1 time(s)
"inar" appears 1 time(s)
"nary" appears 1 time(s)
"sear" appears 1 time(s)
"tree" appears 1 time(s)

4-gram tree is complete: No
4-gram tree is full: No

Total 4-gram count: 10

Total 4-gram count: 13
"aatt" appears 1 time(s)
"arch" appears 1 time(s)
"bina" appears 1 time(s)
"earc" appears 1 time(s)
"eigh" appears 1 time(s)
"heig" appears 1 time(s)
"ight" appears 1 time(s)
"inar" appears 1 time(s)
"nary" appears 1 time(s)
"samp" appears 2 time(s)
"sear" appears 1 time(s)

"tree" appears 1 time(s)
"zinc" appears 1 time(s)

4-gram tree is complete: No
4-gram tree is full: No

Q4) Time Complexities

1.addNgram function

```
void NgramTree::addNgram(string ngram) {
    KeyType word = ngram;
    CountType count = 1;
    TreeItem newItem(word, count);
    insert(root, newItem);
}
```

This function has a recursive function called insert. The time complexity of the addNgram function is equal to the insert function since the other operations done are O(1).

```
void NgramTree::insert(TreeNode*& treePtr, const TreeItem& newItem)
    throw(TreeException) {
    // The item position is found if null has been reached
    if (treePtr == NULL) {
        treePtr = new TreeNode(newItem, NULL, NULL);
        if (treePtr == NULL)
            throw TreeException("TreeException: insert failed");
    }
    // Else search for the insertion position
    else if (newItem.getKey() < treePtr->item.getKey())
        insert(treePtr->leftChildPtr, newItem);
    else if (newItem.getKey() > treePtr->item.getKey())
        insert(treePtr->rightChildPtr, newItem);
    // If the item already exists, increase its count
    else
```

```
                treePtr->item.incrCount();
        }
```

The recurrence relation for the time complexity for insert function in the worst case is

$$T(n) = T(n_{rightchild/leftchild}) + O(1);$$

Where $n_{rightchild/leftchild}$ is the subtree that contains the maximum number of items. In the worst case, the tree looks like a linked list so this value will be equal to n-1.

$$T(n) = T(n-1) + O(1)$$

Using repeated substitutions, we get the following.

$$T(n) = T(n-2) + 2 \cdot O(1)$$
$$T(n) = T(n-3) + 3 \cdot O(1)$$
$$T(n) = T(n-k) + k \cdot O(1)$$

Let k = n - 1 and we get T(n) = T (1) + O(n). Since T(1) = O (1), we find T(n) = O(n) in the worst case. Hence, addNgram function's time complexity is O(n) in the worst case.

2. Operator <<

```
        ostream& operator <<(ostream& out, NgramTree& tree) {
            tree.inorderTraverse();
            return out;
        }
```

The implementation of the << operator here uses inorder traversal.

```
        void NgramTree::inorderTraverse() {
            inorder(root);
        }

        void NgramTree::inorder(TreeNode* treePtr) {
            if (treePtr != NULL) {
                    inorder(treePtr->leftChildPtr);
                    cout << "\"" << treePtr->item.getKey() << "\"" <<"
appears"
   << treePtr->item.getCount() << " time(s)" << endl;
```

```
                        inorder(treePtr->rightChildPtr);
                    }
          }
```

The recurrence relation for the time complexity of inorder is the following in the worst case:

$$
\begin{aligned}
T(n) &= T(n-1) + O(1) \\
T(n) &= T(1) + (n-1) \cdot O(1) \\
T(n) &= O(n)
\end{aligned}
$$

In the equation, we just write T(n) = T (n-1) + O(1) because in the worst case, again, the tree is like a linked list and everything will be done one by one. Since the operation for printing the item is O(1), << operator will be O(n).